CVSD MODULATOR USING VHDL

William T. Hicks, P.E.
&
Robert E. Yantorno, PhD
Temple University

ABSTRACT

IRIG-106 Chapter 5 describes a method for encoding voice using a simple circuit to reduce the overall bit rate and still achieve good quality voice. This well described Continuously Variable Slope Delta Modulation (CVSD) circuit can be obtained using analog parts. A more stable implementation of CVSD can be obtained by designing an anti-aliasing input filter, an A/D converter, and logic. This paper describes one implementation of the CVSD using a standard A/D converter and logic.

KEYWORDS

Voice, Continuously Variable Slope Delta Modulation , CVSD, VHDL

INTRODUCTION

To obtain reasonable quality digitized voice data, the voice signal needs to be sampled at least 8 KSPS with at least 12 bits of resolution. In telephone voice, the 12 bits is reduced to 8 bits by compressing the 12 bit signal with a non-linear compression. This allows a larger overall range of signal levels, by taking advantage of a characteristic of speech that the compression is not noticeable to the listeners. Another characteristic of speech is that the higher frequency signals tend to be at lower signal levels then the low frequency signals. This implies a property similar to what slew limiting does in amplifiers. A method of voice digitization that takes advantage of both of these properties is Continuously Variable Slope Delta Modulation (CVSD) This method can compress telephone quality voice data from a nominal 96 KBPS for simple encoding to 16 KBPS for CVSD encoding, with similar quality.[3]

BACKGROUND

Delta Modulation is simply the one bit discrete output of a comparator that indicates if an estimate of the signal is currently less than or greater than the signal being digitized. In Figure 1, the input

signal is compared to an integrated version of the sampled output signal. A sample of the input and output of this modulator is shown in Figure 2. Note how the feedback signal never catches up to the input signal until the input signal changes direction, assuming the input signal is of a high enough slew rate. [3]

A method to overcome this slew problem is the CVSD modulator shown in Figure 3. In this figure, a new block (the "syllable integrator") has been added. This function recognizes when the signal integrator is not catching up to the input signal fast enough, by monitoring the digital output to find when the output is not changing state for a preset number of times. When this occurs, the syllable integrator increases the signal level into the signal integrator. As long as the output of the comparator stays in the same state, the syllable integrator output keeps increasing in magnitude. This increases the speed at which the signal integrator reaches the input signal level, as shown in Figure 4. Once the signal reaches the input signal level, the digital output reverses and the syllable integrator starts to reduce its level. Note, that if the digital output is inverted, the only effect in the receiver (decoder) will be to invert the polarity of the analog output signal, which will not be noticed by a listener. [3]

The block diagram shown in Figures 3 is a classic analog implementation of the CVSD encoder. A digital approach is shown in Figure 5. Here the two integrators are implemented in digital logic, and the feedback is a 10 bit Digital to Analog (D/A) converter. This approach provides a more reproducible converter, as the integration times are only a function of the clock rate, with the integrators implemented with digital arithmetic functions.[1] The Harris IC [2] that uses this implementation is no longer in production. Many of the voice digitizing systems already have a 12 bit Analog to Digital encoder (A/D) on them, capable of running at 16 KSPS, and spare logic in a Field Programmable Gate Array (FPGA). This paper describes an implementation of the CVSD function using those parts.

IMPLEMENTATION

Figure 6 shows an implementation of the CVSD encoder function using digital logic and an A/D converter. The parts of this implementation are: 1) the A/D converter, 2) the input signal to feedback signal comparator, 3) the overload shift register, 4) the syllable filter integrator, and 5) the signal integrator. Note that the system runs off of a 16 KHz clock, which meets one of the requirement of IRIG-106 Chapter 5 [4].

Analog to Digital Converter.
This function is a simple 12 bit parallel converter that runs off the 16 KHz sampling clock. The A/D converter outputs a signal from zero for negative full scale to all ones for positive full scale. Inverting the msb will change the output to two's complement. The A/D, in two's complement (1.11 format), outputs the negative full scale signal as a –1 and the positive full scale signal as + (1-lsb). 12 bits is probably more than needed, but the A/D converter was readily available. The two lsb's could probably be deleted and not cause a discernable change in the performance.

The A/D input goes through a DC bias correction circuit. This circuit consists of a counter that counts up or down, depending on the level of the serial output. The upper 8 bits of the 16 bit counter

are added into the lower 8 bits (sign extended) of the A/D, allowing for a slow correction of DC offset error in the A/D converter. This results in the A/D bias being reduced to the point that the serial output with no AC signal in is an alternating 101010 pattern. The correction rate is below the pass band low frequency corner of the input analog band pass filter.

Comparator.
This function compares the signed magnitude of the input to the feedback signal. When using VHDL "standard logic vector's" for the comparator, the input and feedback must be in a zero to all ones signal basis. This was accomplished by using the A/D signal unmodified and the feedback signal with the msb inverted. Because the arithmetic circuits that generated the feedback signal use many more bits than the A/D, the feedback signal is truncated to match the length of the A/D.

Shift register and over run detector.
A simple two stage shift register was implemented. The data signal and the two outputs of the shift register are feed into a function that produces a "one" out if the three inputs are the same. If the output of this circuit is a zero, then the lower value of syllable signal is used, otherwise the higher value is used.

Syllable level selection.
Initially the values used were taken from the Harris part and were 1/16 for the higher level and 1/512 (1/505) for the lower level. [1] Simulations were run and it was found that these values met the IRIG spec for 24 dB compression, however, the nominal signal level for the "30% run of threes" test was very low compared to the full signal range of the A/D. While this point can be set over a broad range, it was felt that a more useful range would be higher and the values were increased by four times. Simulations were rerun and full signal level at the 800 Hz test frequency were now at 25% of the A/D limits. Both of these amplitudes are pre-multiplied by the value of "1-a" so that the output of the syllable integrator tends to move toward the desired amplitude over time (see below). For the original Harris values, this produces a signal of 16 bits long, much longer than the 12 bit A/D signal.

Integrators.
A simple digital integrator (see Figure 7) just delays a two's complement signal by one clock and then adds it to the next input (a = 1). This is a true integrator, and will move toward an infinite output, or a saturated output with fixed length words. This can be modified by using only a portion of the input as the delayed feedback. With the modified feedback($a<1$), the output will reach a limit of $1/(1-a)$ times the input.

Syllable Integrator.
The syllable integrator is required to have a time constant of 4 to 6 milliseconds (mS). A value of 4 mS was selected. With a sample rate of 16 KHz, 4 mS yields 64 samples. Therefore "a" should be $63/64 = 0.984375$. This was selected so that the multiplications could be performed by shifting. Multiplying by 63/64 is the same as multiplying by (1-1/64), which only requires shifting the input by six bits and subtracting, thereby simplifying the logic needed in the multiplication. Note that the number of bits needed increases by 6.

Modulator.
The modulator is a circuit that simply multiplies the value of the output of the syllable integrator times +1 if the feedback signal needs to be increased and by –1 if the signal needs to be decreased.

Output Integrator.
The output integrator is required to have a time constant of 0.75 to 1.25 mS. By selecting a time constant of 1 mS, the 16 KHz system provides 16 samples for the 1 mS. This gives a value of "b" of 15/16 = 0.9375. As described before, the circuit can be implemented by using (1-1/16) as the multiplier, and simplifying the logic. Note that the number of bits needed increases by 4 bits. This integrator is susceptible to overflow, so logic was added to sense overflow and saturate the output.

Bit Growth.
If the multiplication function was implemented with standard multipliers, there would be bit growth of two times at each stage. This would result in the final value being 64 bits long. While the shift function reduces this, it does not eliminate it, and there is some bit growth. The output of the syllable integrator is 22 bits long and the output of the signal integrator is 26 bits long. This 26 bit signal was truncated to compare to the 12 bit A/D signal.

VHDL CODE

The VHDL code (see the appendix) was written for a XILINX FPGA, however it should be generic enough to be able to use with any FPGA. The code was simulated to verify operation, however it was not implemented in a working system.

CONCLUSIONS

The implementation of a CVSD encoder using VHDL code has been shown to be relatively easy to implement. While the decoder is not shown, it should be easy to implement using portions of the encoder code. The methods of selecting the values of the multipliers have been explained to allow the calculation of values for other sample frequencies and A/D converter bit lengths. Overall the code generated should be easily modified to meet most requirements. While the use of straight forward multipliers could be implemented, the problem with bit growth of their outputs was not addressed.

REFERENCES

[1] Donovan, Dave, "An Algorithmic Representation of CVSD", Harris Semiconductor report.

[2] Harris Semiconductor, "Continuously Variable Slope Delta-Modulator (CVSD)", Data Sheet # HC-55564/883, June 1994.

[3] Harris Semiconductor, "Delta Modulation For Voice Transmission", Application Note # 607.1, January 1997.

[4] Range Commanders Council, “Telemetry Standards”, IRIG-106-01, Part I, 2001

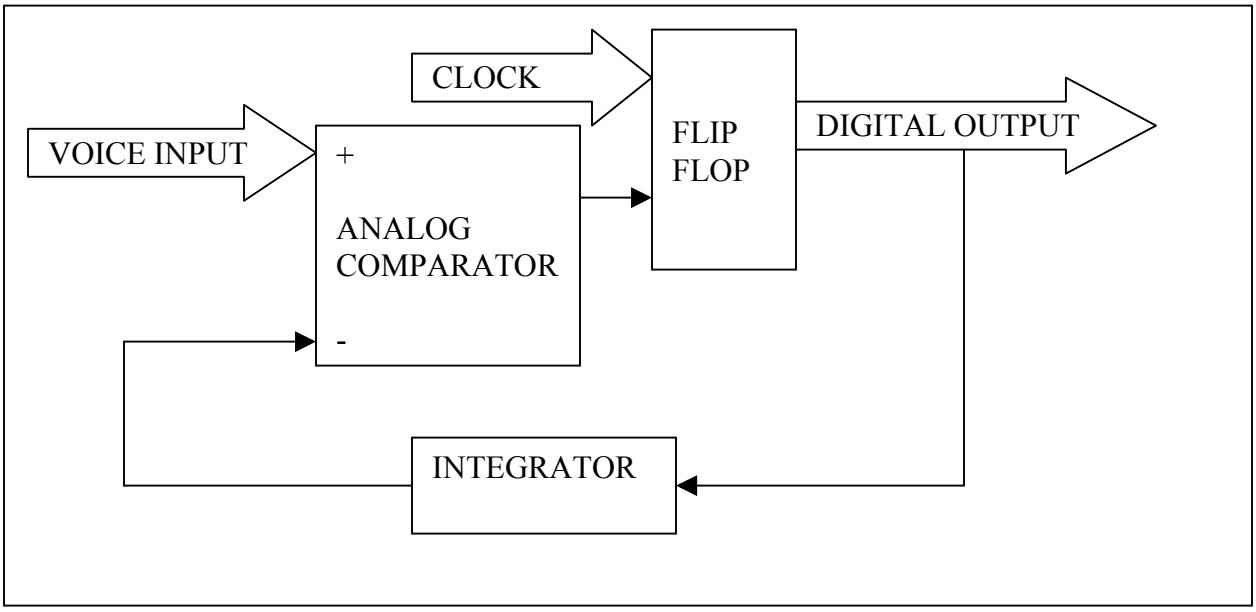

FIGURE 1 Delta Modulator.

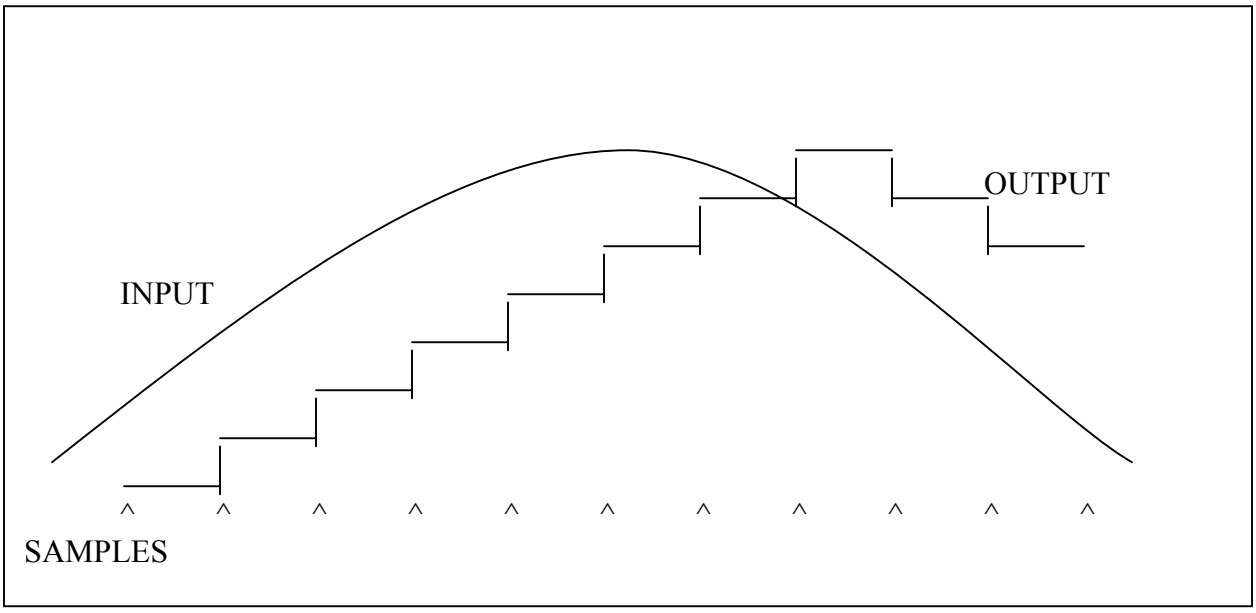

FIGURE 2 Delta Modulation.

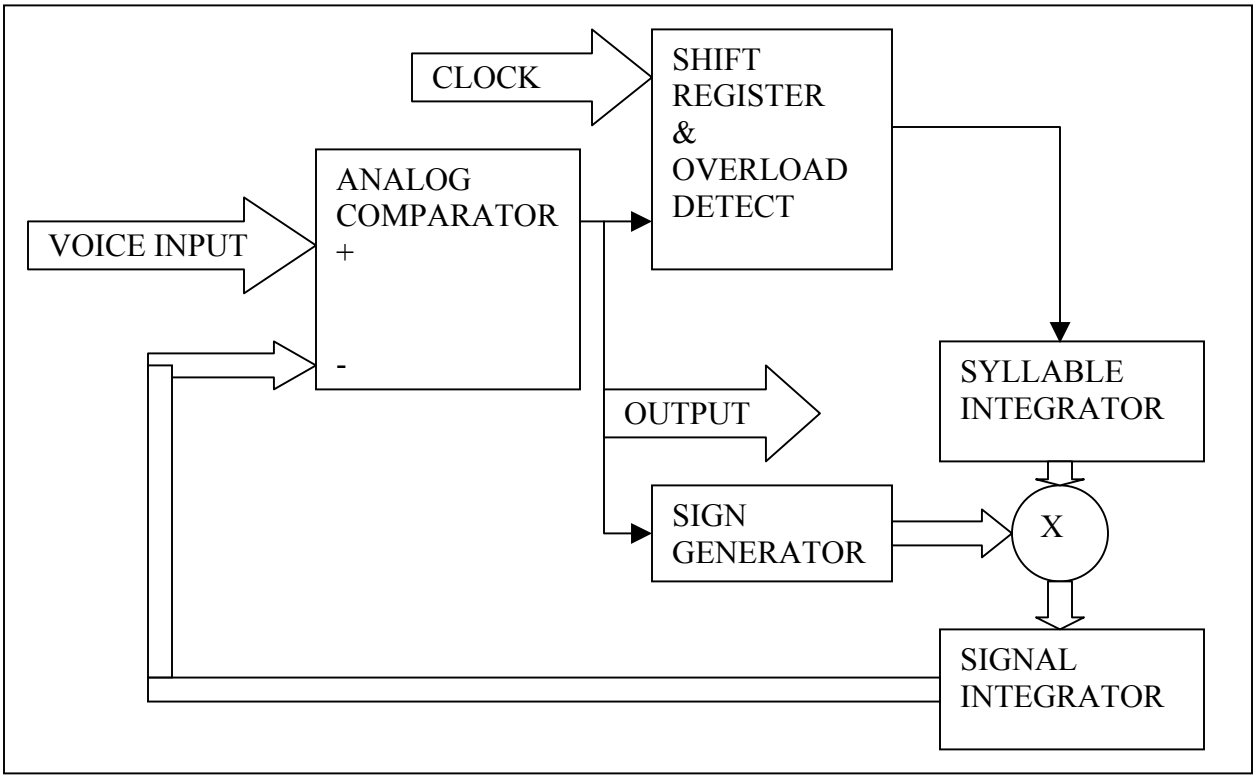

FIGURE 3 Continuously Variable Slope Delta Modulator.

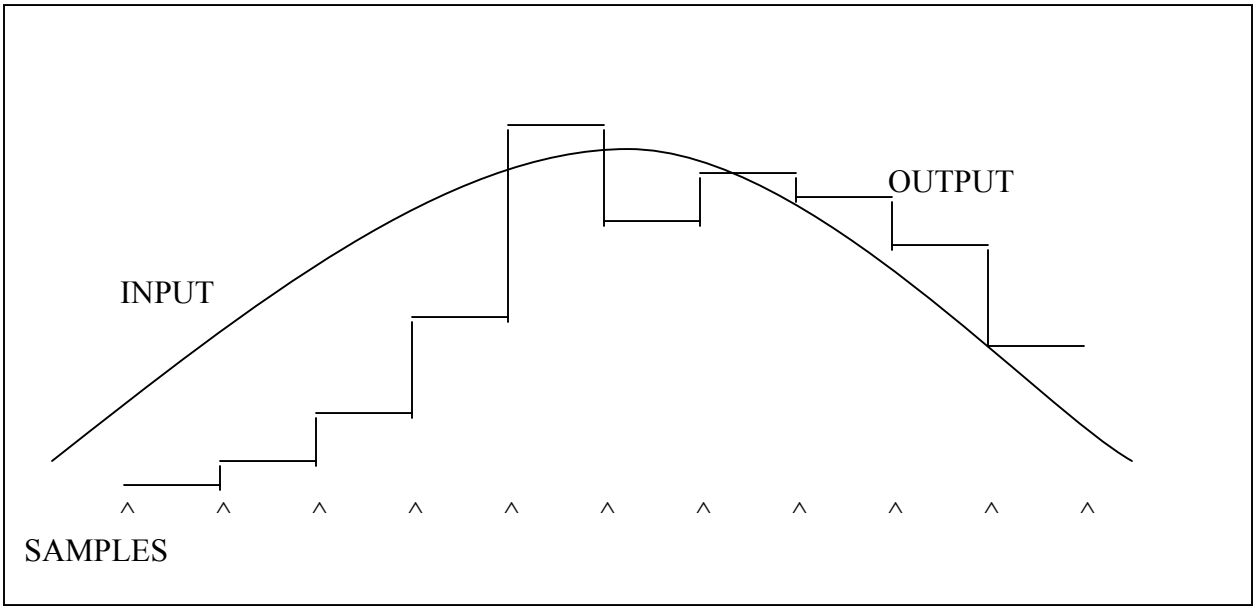

Figure 4 Continuously Variable Slope Delta Modulation.

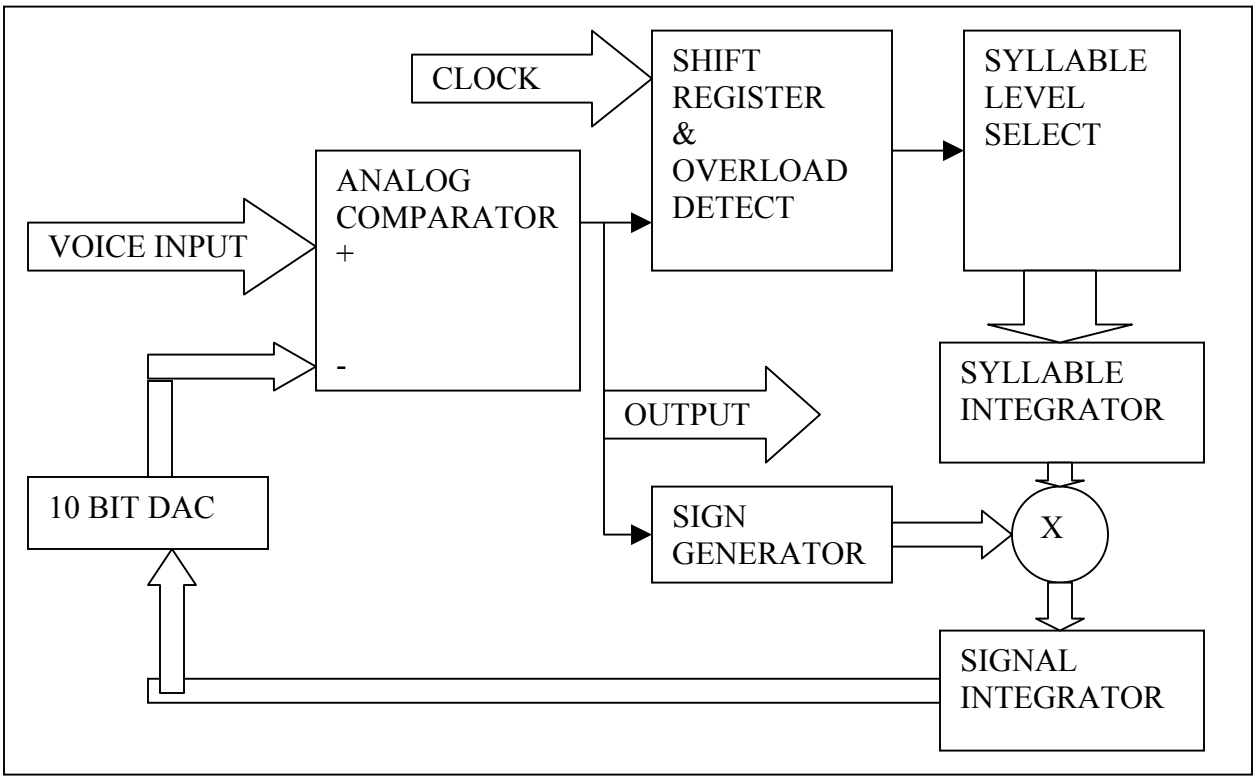

FIGURE 5 Continuously Variable Slope Delta Modulator with DAC.

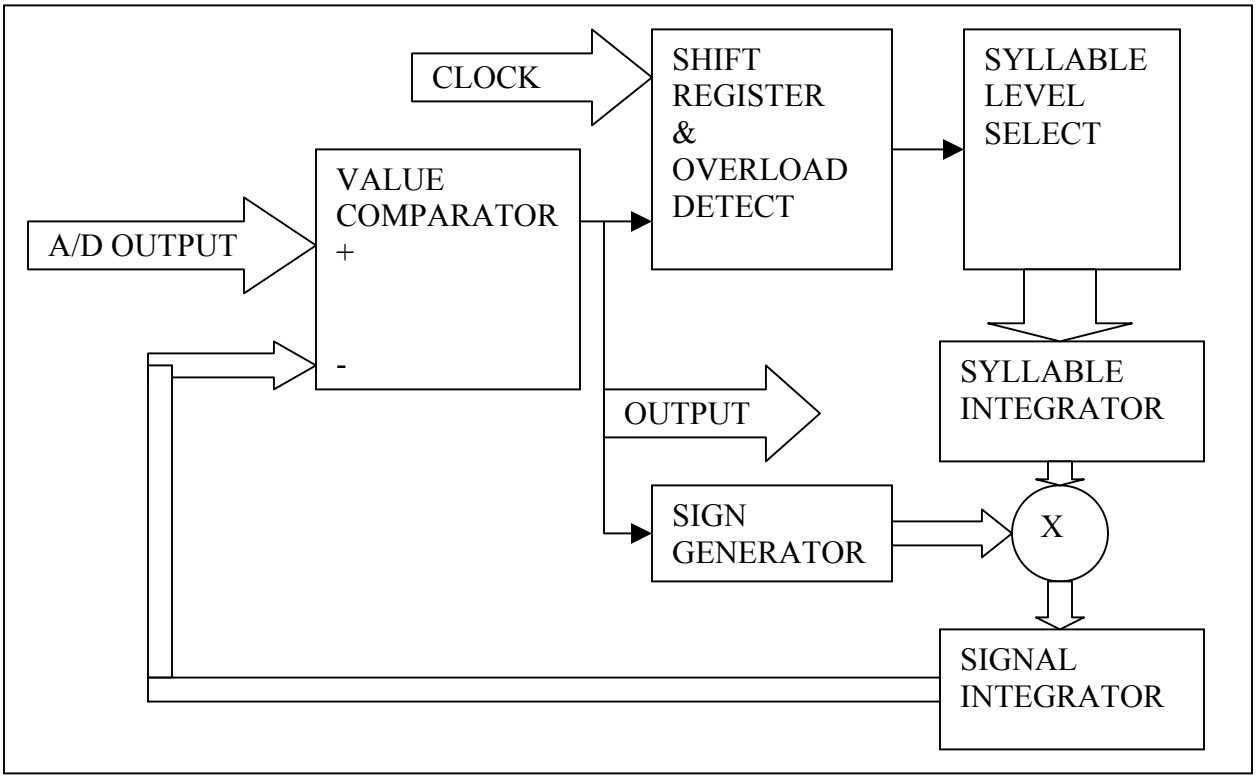

FIGURE 6 Continuously Variable Slope Delta Modulator with A/D.

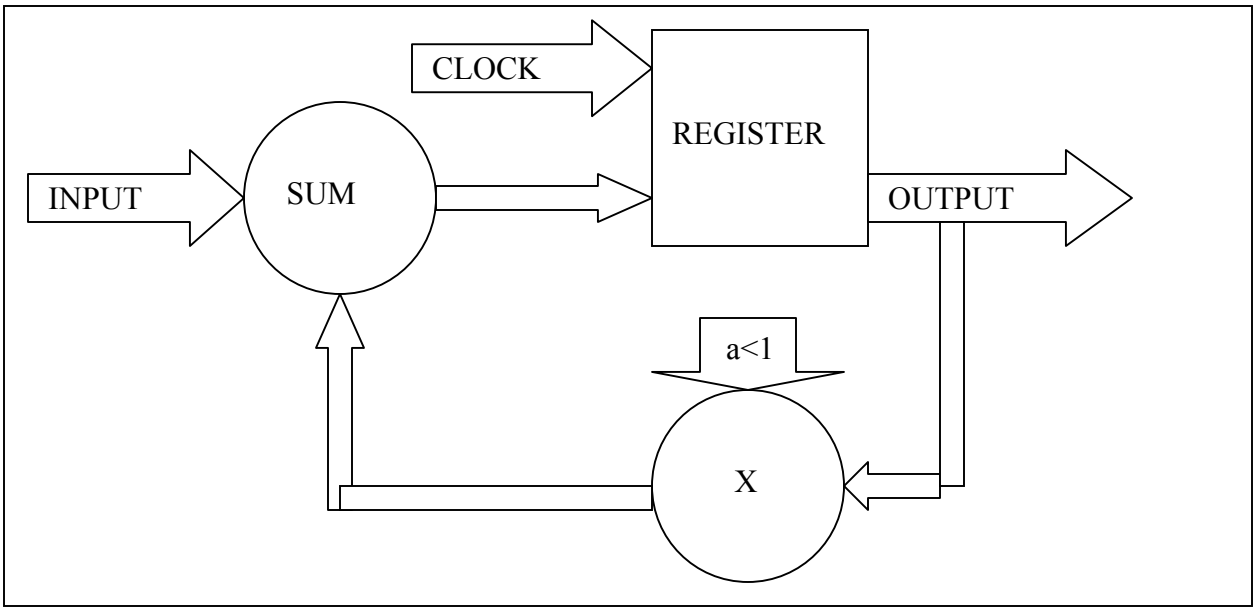

FIGURE 7 Integrator.

APPENDIX

```
--------------------------------------------------------------------------------------------
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
--------------------------------------------------------------------------------------------
entity CVSD is
       port (
              CLK:           in STD_LOGIC;          --input clock, 16 KHz
              DECODE:        in STD_LOGIC;                         --set high to activate decoder
              A_D_DATA:      in STD_LOGIC_VECTOR(11 downto 0);     --output of A/D converter
              SIGNED_INPUT:  out STD_LOGIC_VECTOR(11 downto 0);    --signed output of A/D converter
              SIGNED_OUTPUT: out STD_LOGIC_VECTOR(11 downto 0);    --signed output of feedback
              UNSIGNED_OUTPUT:out STD_LOGIC_VECTOR(11 downto 0);   --unsigned output of feedback
              SERIAL_IN:     in STD_LOGIC;          --input to decoder
              SERIAL_OUT:    out STD_LOGIC          --clocked output
              );
end entity CVSD;
--------------------------------------------------------------------------------------------
architecture inside of CVSD is
signal  INPUT:                        STD_LOGIC_VECTOR(11 downto 0):=X"000";
signal  COUNT:                        STD_LOGIC_VECTOR(15 downto 0):=X"0000";
signal  DATA:                         STD_LOGIC_VECTOR(2 downto 0):="000";
signal  DATAx:                        STD_LOGIC:='0';
signal  SYLLABLE:                     STD_LOGIC_VECTOR(15 downto 0):=X"0000";
signal  SYLLABLE_SUM:                 STD_LOGIC_VECTOR(21 downto 0):="00"&X"00000";
signal  SYLLABLE_SUM_DELAY:           STD_LOGIC_VECTOR(21 downto 0):="00"&X"00000";
signal  OUTPUT:                       STD_LOGIC_VECTOR(21 downto 0):="00"&X"00000";
signal  FEEDBACK_DATA:                STD_LOGIC_VECTOR(25 downto 0):="00"&X"000000";
signal  FEEDBACK_DATAx:               STD_LOGIC_VECTOR(26 downto 0):="000"&X"000000";
signal  ONE_MINUS_A_SYLLABLE_SUM:     STD_LOGIC_VECTOR(21 downto 0):="00"&X"00000";
signal  ONE_MINUS_B_FEEDBACK_DATA:    STD_LOGIC_VECTOR(25 downto 0):="00"&X"000000";
constant       MINIMUM:               STD_LOGIC_VECTOR(15 downto 0):=X"0004";--1/128*1/64
constant       MAXIMUM:               STD_LOGIC_VECTOR(15 downto 0):=X"0080";--1/4*1/64
constant       ZERO:                  STD_LOGIC_VECTOR(21 downto 0):="00"&X"00000";
begin --architecture
BUFFER_IN: process (CLK) -- Buffer input to make synchronous to rest of circuit
begin
       if rising_edge (CLK) then
              INPUT <= A_D_DATA + (COUNT(15) & COUNT(15) & COUNT(15) & COUNT(15) &
               COUNT(15 downto 8));
       end if;
end process BUFFER_IN;
--
RESTORE_BIAS: process (CLK)
begin
       if rising_edge (CLK) then
              if DATA(1) = '1' then
                     COUNT <= COUNT + 1;
              else
                     COUNT <= COUNT - 1;
              end if;
       end if;
end process RESTORE_BIAS;
--
COMPARE: process (INPUT,FEEDBACK_DATA,DATAx,DECODE)
       -- Compare magnitude of input and 1's complement of output to find greater
begin
       if DECODE = '1' then
              DATA(0) <= DATAx;             --used to test decoder
       elsif ((not FEEDBACK_DATA(25)) & FEEDBACK_DATA(24 downto 14)) > INPUT then
              DATA(0) <= '1';
       else
              DATA(0) <= '0';
       end if;
end process COMPARE;
--
DELAY: process (CLK) -- Buffer input to make synchronous to rest of circuit
```

```
begin
	if rising_edge (CLK) then
		DATAx <= SERIAL_IN;		--used to test decoder
		DATA(1) <= DATA(0);
		DATA(2) <= DATA(1);
	end if;
end process DELAY;
--
SYLLABLE_MAGNITUDE: process (DATA) -- Set syllable magnitude based on DATA history
	-- Number is pre multiplied by (1-a)
begin
	if DATA = "000" or DATA = "111" then
		SYLLABLE <= MAXIMUM;
	else
		SYLLABLE <= MINIMUM;
	end if;
end process SYLLABLE_MAGNITUDE;
--
SYLLABLE_SUM <= (SYLLABLE & "000000") + SYLLABLE_SUM_DELAY; --integrate
--
ONE_MINUS_A_SYLLABLE_SUM <= SYLLABLE_SUM(21) & SYLLABLE_SUM(21)
 & SYLLABLE_SUM(21) & SYLLABLE_SUM(21) & SYLLABLE_SUM(21)
  & SYLLABLE_SUM(21) & SYLLABLE_SUM(21 downto 6);--arithmetic shift 6
--
DELAY_2: process (CLK) --delay and multiply by A
begin
	if rising_edge (CLK) then
		SYLLABLE_SUM_DELAY <= SYLLABLE_SUM - ONE_MINUS_A_SYLLABLE_SUM;
	end if;
end process DELAY_2;
--
OUTPUT_MAGNITUDE: process (DATA(0),SYLLABLE_SUM) -- Set magnitude sign based on DATA
begin
	if DATA(0) = '0' then
		OUTPUT <= SYLLABLE_SUM;
	else
		OUTPUT <= ZERO - SYLLABLE_SUM;
	end if;
end process OUTPUT_MAGNITUDE;
--
ONE_MINUS_B_FEEDBACK_DATA <= FEEDBACK_DATA(25) & FEEDBACK_DATA(25) & FEEDBACK_DATA(25)
 & FEEDBACK_DATA(25) & FEEDBACK_DATA(25 downto 4);--arithmetic shift by 4
--
DELAY_3: process (CLK) --integrate output
begin
	if rising_edge (CLK) then
		FEEDBACK_DATAx <= (FEEDBACK_DATA(25) & FEEDBACK_DATA) -
		(ONE_MINUS_B_FEEDBACK_DATA(25) & ONE_MINUS_B_FEEDBACK_DATA) +
		(OUTPUT(21) & OUTPUT & "0000");
	end if;
end process DELAY_3;
--
SATURATE: process (FEEDBACK_DATAx)
	--test and fix overflow
begin
		if FEEDBACK_DATAx(26 downto 25) = "00" or
				FEEDBACK_DATAx(26 downto 25) = "11" then	--not over/under flow
			FEEDBACK_DATA <= FEEDBACK_DATAx(25 downto 0);
		elsif FEEDBACK_DATAx(26) = '0' then
			FEEDBACK_DATA <= "01111111111111111111111111";--saturate for overflow
		else
			FEEDBACK_DATA <= "10000000000000000000000000";--saturate for underflow
		end if;
end process SATURATE;
--
SERIAL_OUT <= DATA(0);
SIGNED_INPUT <= (not INPUT(11)) & INPUT(10 downto 0);
SIGNED_OUTPUT <= FEEDBACK_DATA(25 downto 14);
UNSIGNED_OUTPUT <= (not FEEDBACK_DATA(25)) & FEEDBACK_DATA(24 downto 14);
end architecture inside;
--------------------------------------------------------------------------------
```